Introduction to Deep Learning

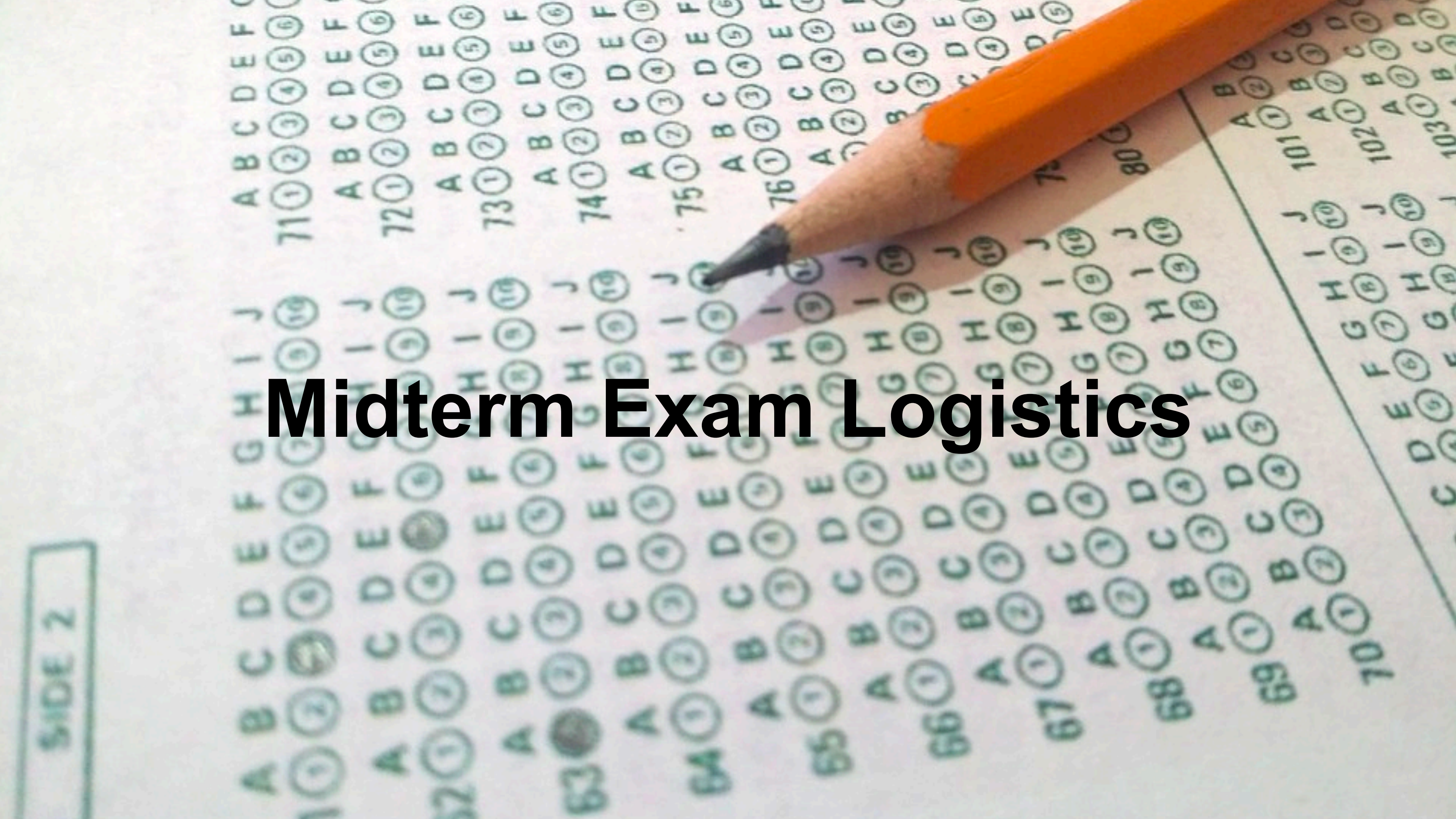
14. Hybridize, Async, Multi-GPU/Machine Training

STAT 157, Spring 2019, UC Berkeley

Alex Smola and Mu Li

courses.d2l.ai/berkeley-stat-157

Midterm Exam Logistics



Logistics

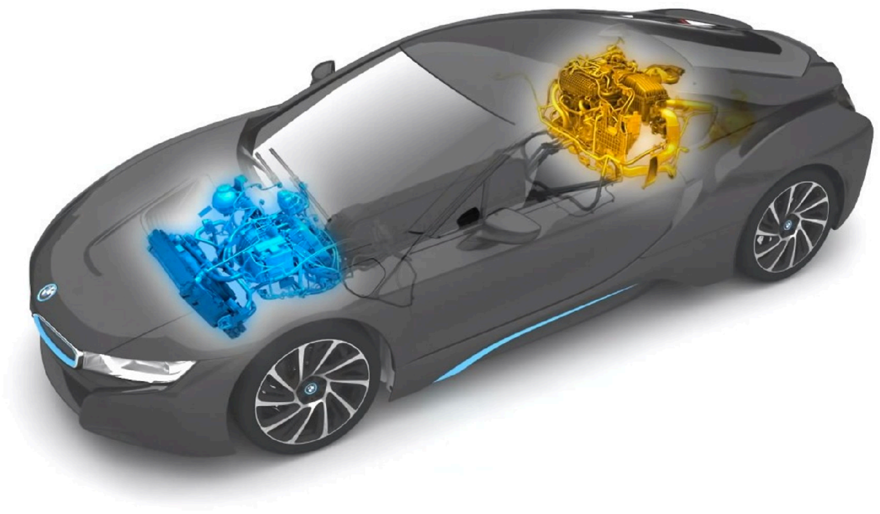
- Tuesday, March 19, 2019, 3:30pm until 5:00pm
- LeConte 1 and 3 (we will fill #1 first)

- Open Book but no computers / phones / electronics
- Note paper all provided
- You can bring as much paper as you want, printed, written, … (e.g. lecture slides, bookchapters, books).

Questions

- Everything we covered up to the midterm
  - Concepts (e.g. learning rates, regularization)
  - Code (e.g. read a network definition)
  - Math (e.g. taking derivatives)
  - Experiments (e.g. debug logs from training a net)
- The answers can be simple.
- There will be more questions than you have time (80 minutes)

A Hybrid of Imperative and Symbolic Programming



Imperative Programming

- What we used so far
- The common way to program in Python, Java, C/C++, …
- Straightforward, easy to debug
- Always needs the Python interpreter
    - Hard to deploy models, e.g. smart phones
- Performance issue

Interpreter compiles into bytecode, then execute on virtual machine

```
a = 1
b = 2
c = a + b
```

3 calls in total

Symbolic Programming

- Define the program first, feed with data to execute later
- Math, SQL, …
- Easy to optimize, less frontend overhead, portable
- Hard to use

Know the whole program, easy to optimize

```
expr = "c = a + b"
exec = compile(expr)
exec(a=1, b=2)
```

May be used without Python interpreter

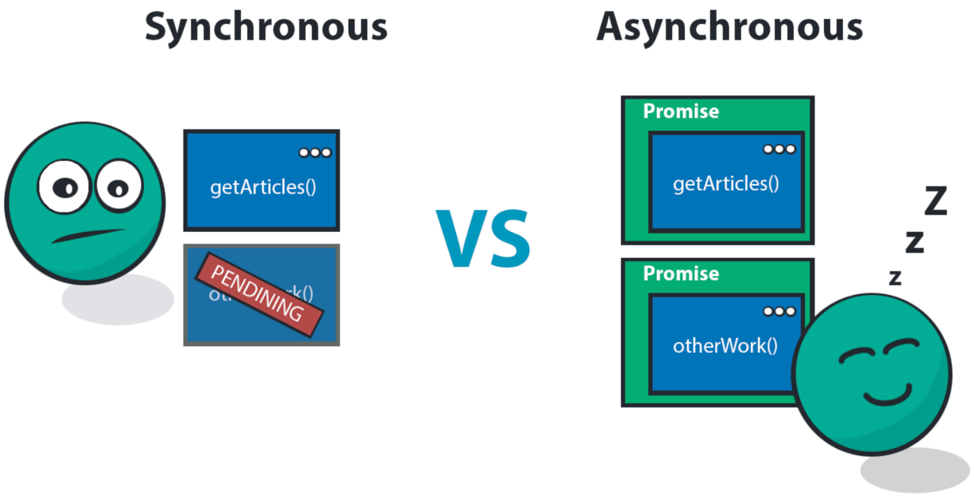
Single call

Hybridization in Gluon

- Define a model through nn.HybridSequential or nn.HybridBlock
- Call .hybridize() to switch from imperative execution to symbolic execution

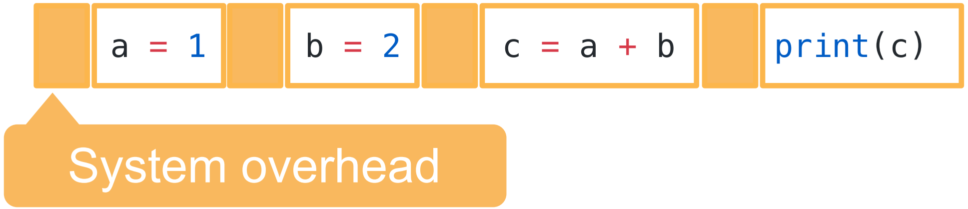
```
net = nn.HybridSequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
net.hybridize()
```

Asynchronous Computing



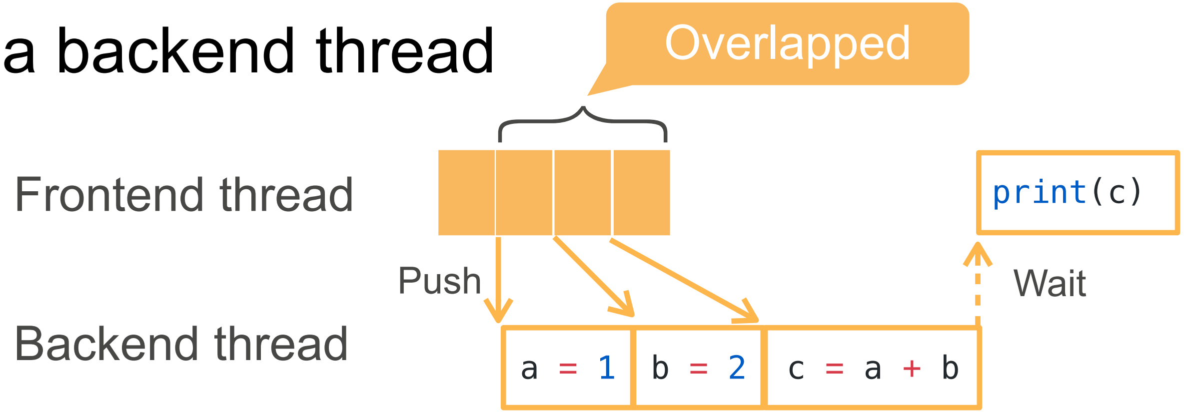
Asynchronous Execution

- Execute one-by-one



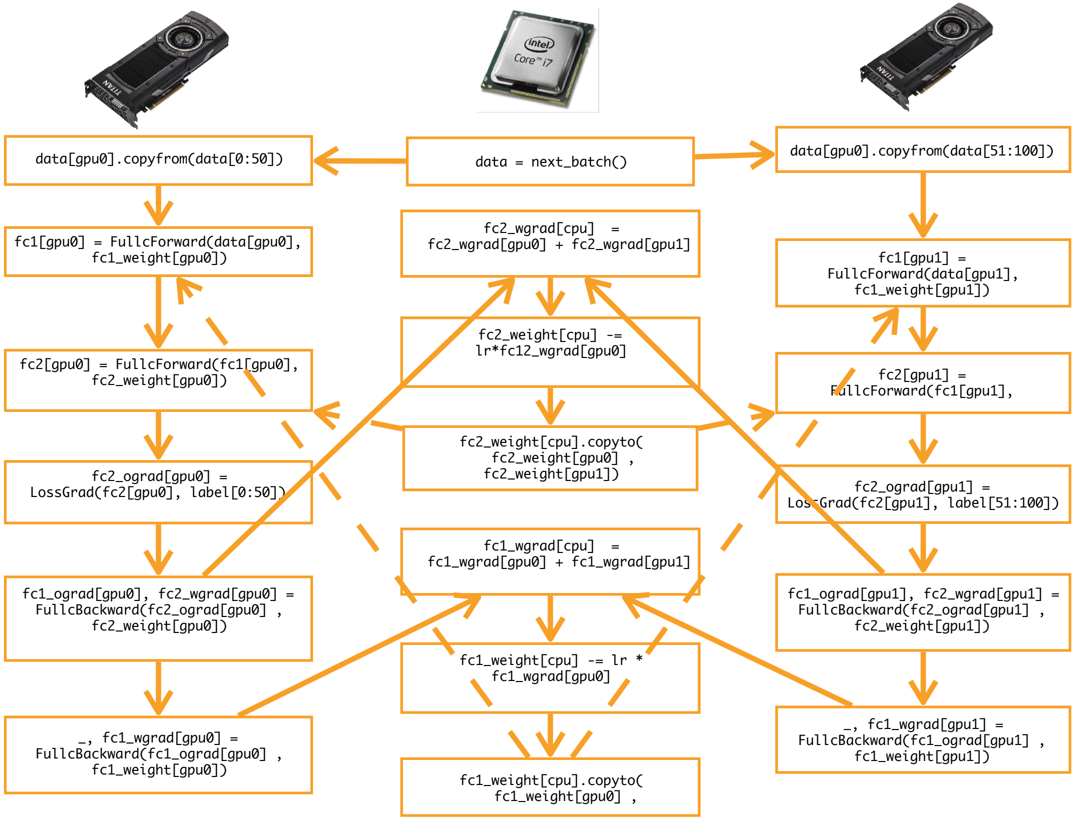

```
a = 1
b = 2
c = a + b
print(c)
```

- With a backend thread



Automatic Parallelism

Writing Parallel Program is Painful



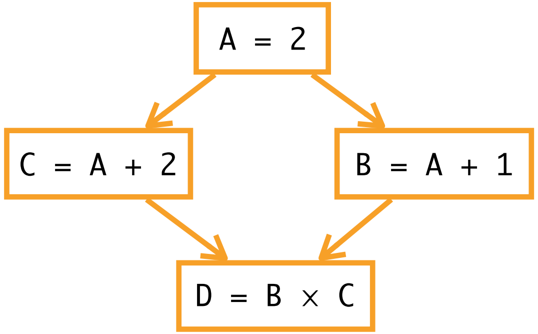
- Single hidden-layer MLP with 2 GPUs
- Scales to hundreds of layers and tens of GPUs

Auto Parallelization

Write serial programs

```
A = nd.ones((2,2)) * 2
C = A + 2
B = A + 1
D = B * C
```

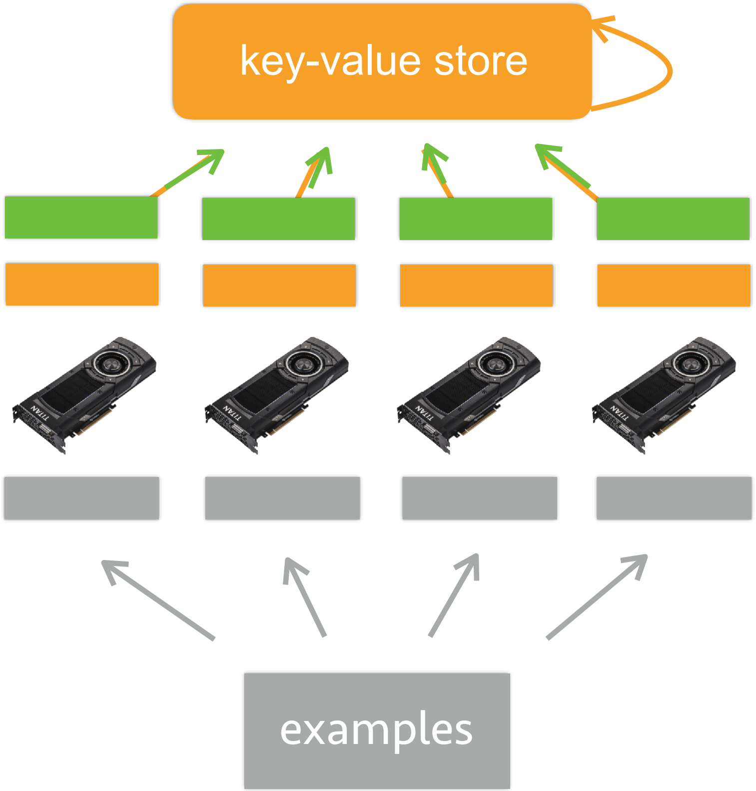
Run in parallel



Multi-GPU Training

(Lunar new year, 2014)

Data Parallelism

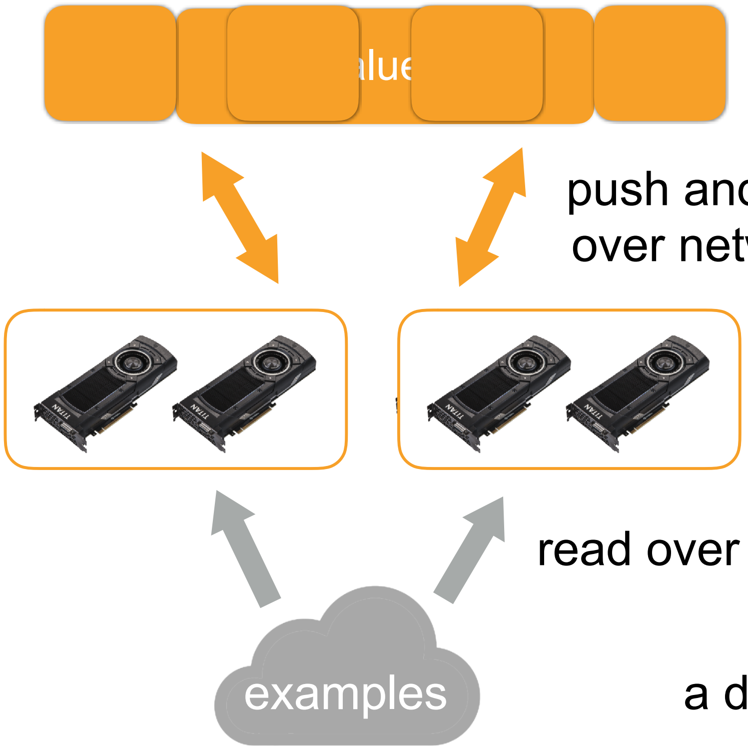


1. Read a data partition
2. Pull the parameters
3. Compute the gradient
4. Push the gradient
5. Update the parameters

Distributed Training

(Alex’s frugal GPU cluster at CMU, 2015)

Distributed Computing



multiple
server machines

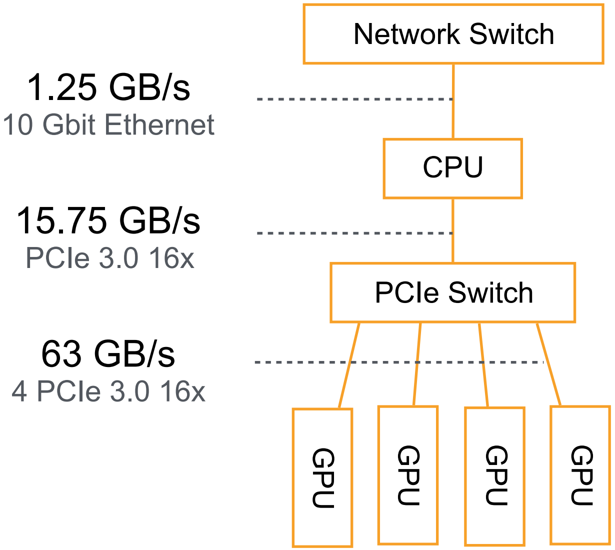
push and pull
over network

multiple
worker machines

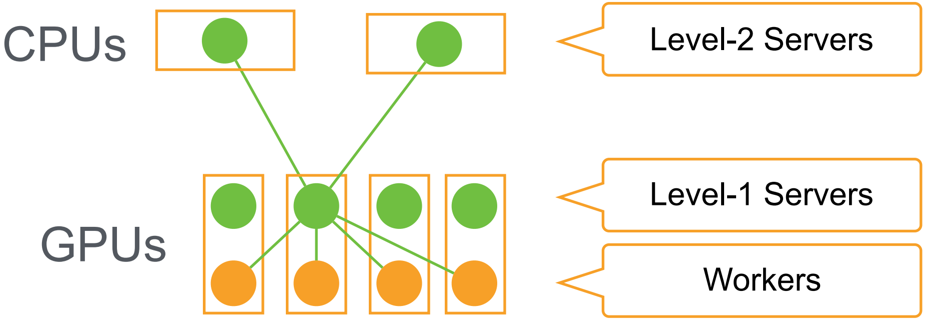
read over network

Store data in
a distributed filesystem

GPU Machine Hierarchy

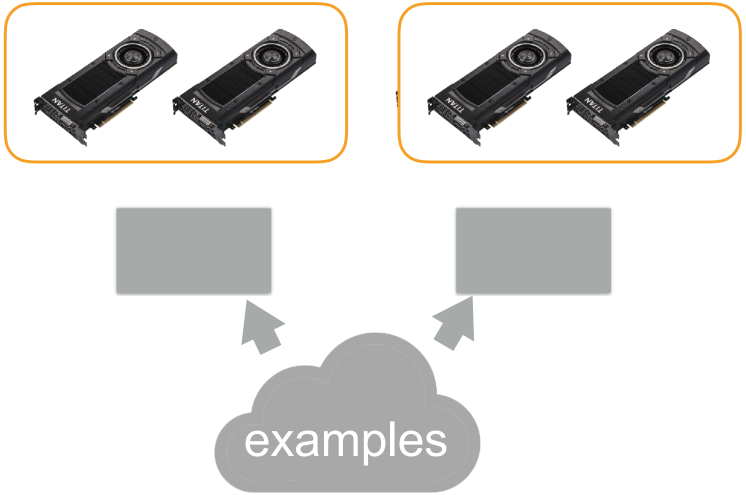


Hierarchical parameter server



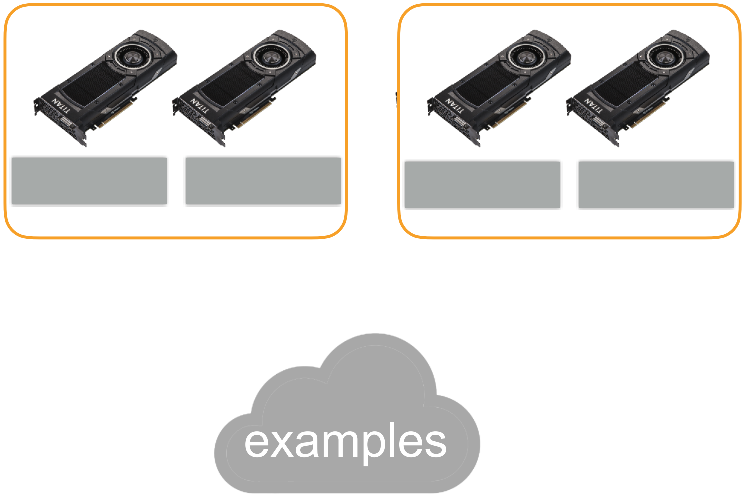
Iterating a Batch

- Each worker machine read a part of the data batch

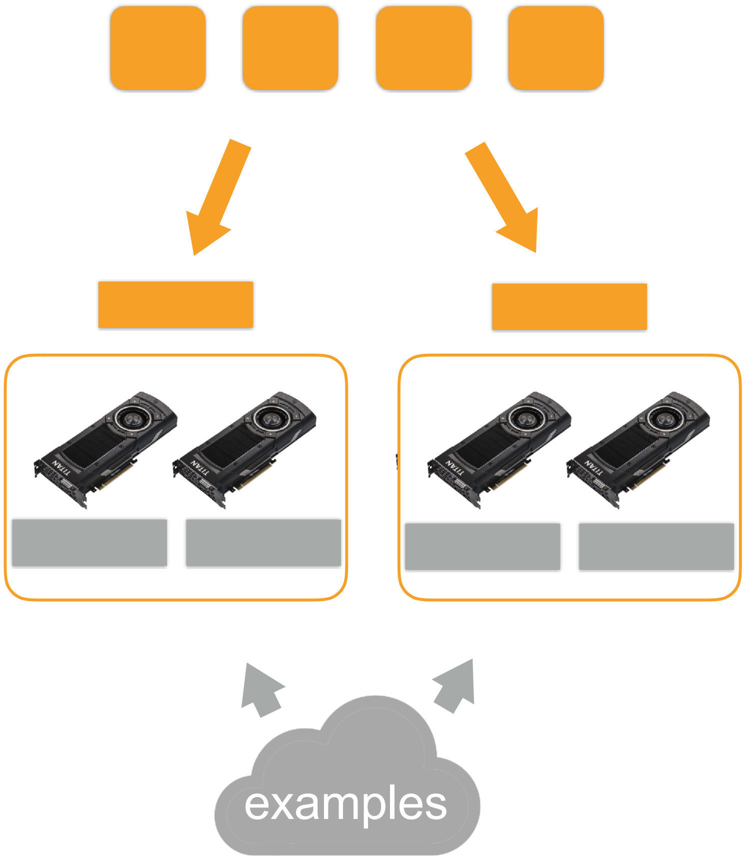


Iterating a Batch

- Further split and move to each GPU

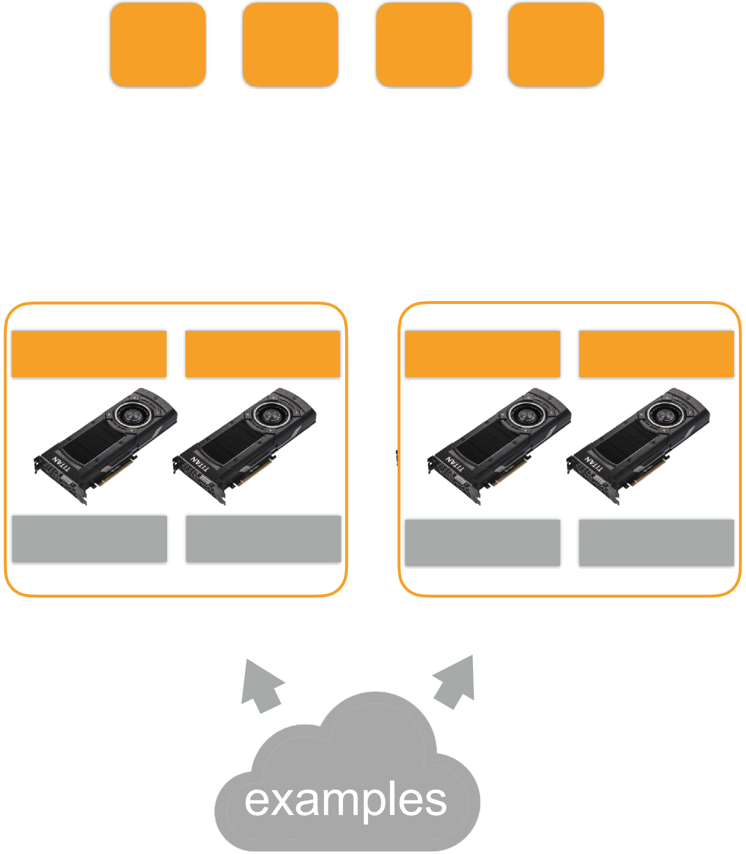


Iterating a Batch



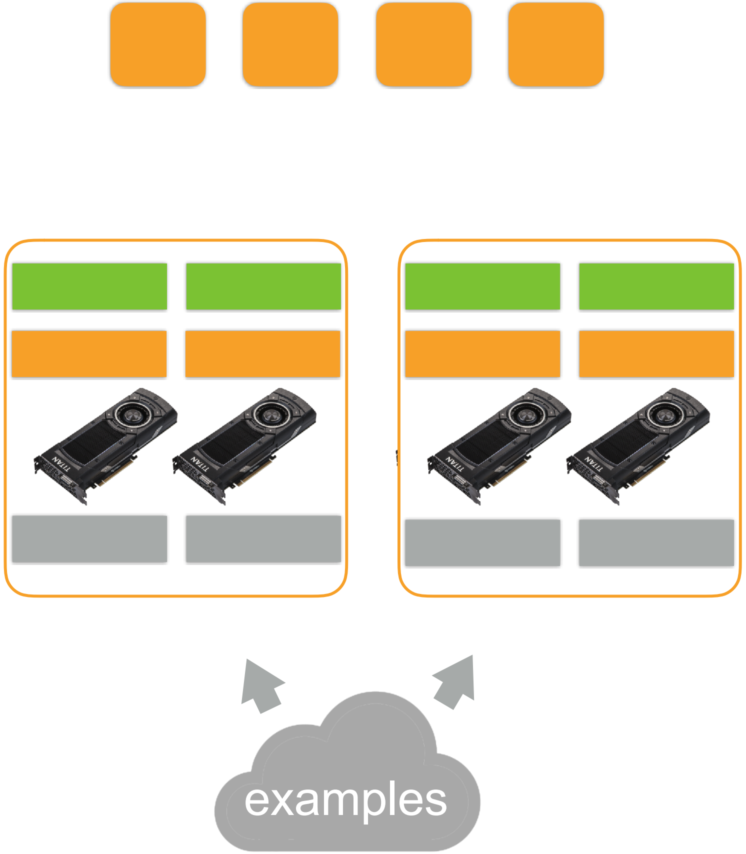
- Each server maintain a part of parameters
- Each worker pull the whole parameters from servers

Iterating a Batch



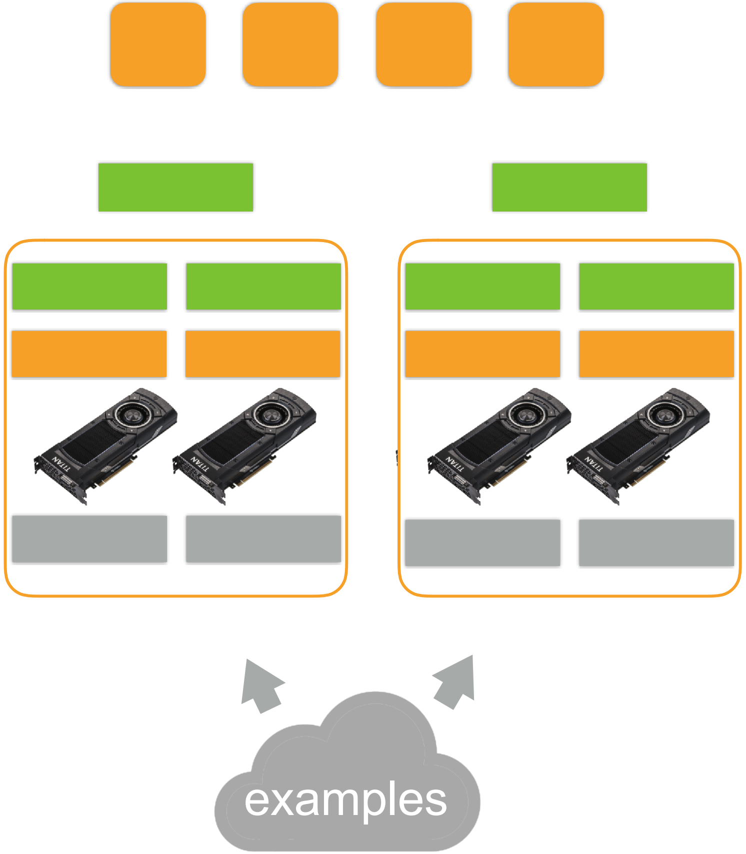
- Copy parameters into each GPU

Iterating a Batch



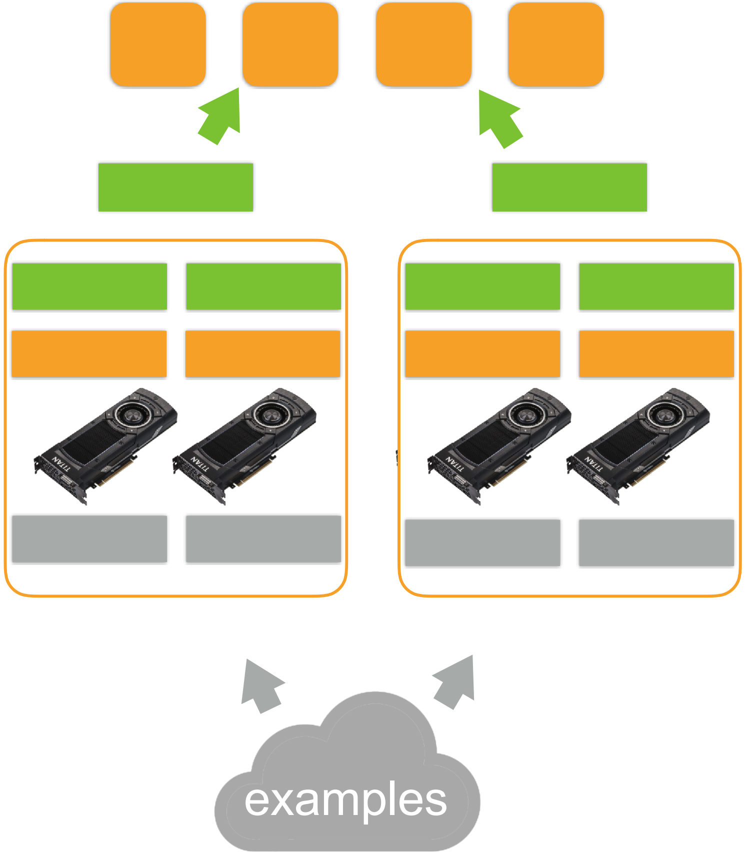
- Each GPU computes gradients

Iterating a Batch



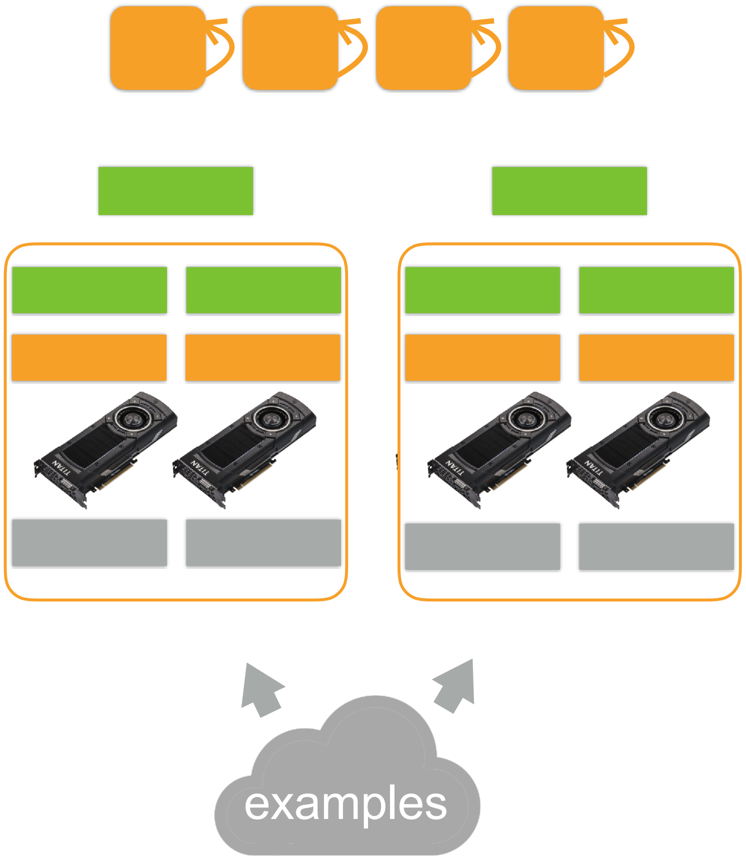
- Sum the gradients over all GPU

Iterating a Batch



- Push gradients into servers

Iterating a Batch



- Each server sum gradients from all workers, then updates its parameters

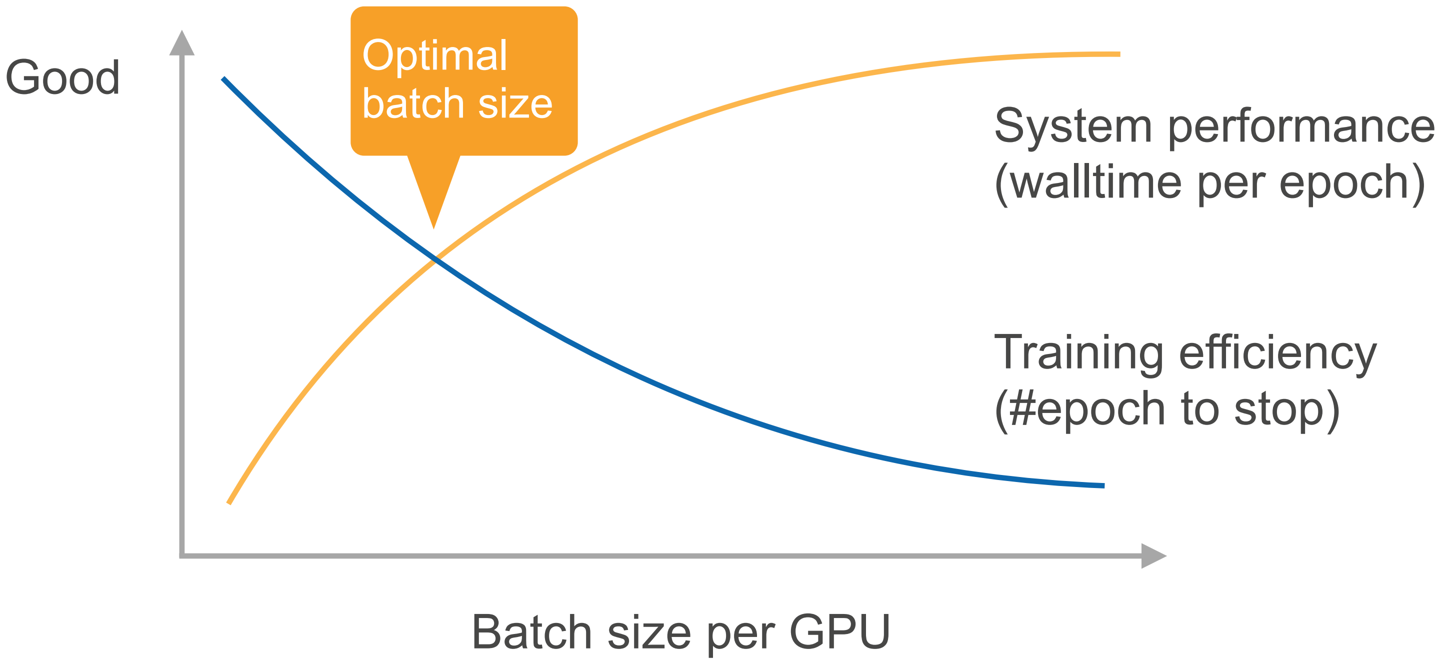
Synchronized SGD

- Each worker run synchronically, called synchronized SGD
- Assume *n* GPUs and each GPU process *b* examples per time, synchronized SGD equals to mini-batch SGD on a single GPU with *nb* batch size
- In the ideal case, training with *n* GPUs will lead to a n times speedup compared to a single GPU training

Performance

- T1 = time to compute gradients for *b* example in a GPU
- Assume *m* parameters, a worker sends and receive m parameters/gradients each time
  - T2 = time to send and receive
- Wall-time for each batch = max(T1, T2)
  - Need to choose large enough *b*
- Increasing *b* and/or *n* leads to a larger batch size, which may need to more data epochs to reach a desired model quality

Performance Trade-off



Practical Suggestions

- A large dataset
- Good GPU-GPU and machine-machine bandwidth
- Efficient data loading/preprocessing
- A model with good computation (FLOP) vs communication (model size) ratio
    - Inception > ResNet > AlexNet
- A large enough batch size for good system performance
- Tricks for efficiency optimization with a large batch size